Cake: a tool for adaptation of object code

Stephen Kell
Stephen.Kell@cl.cam.ac.uk

Computer Laboratory

University of Cambridge
Some familiar problems

Software is

■ expensive to develop
■ expensive to maintain
■ inflexible
Some common ideas, all entailing mismatch

Better programming languages
- great for new codebases
- **mismatch**: inevitably *many* languages

Decentralised development
- many variant codebases evolving in parallel
- **mismatch**: no more interface consensus

Unanticipated composition
- **mismatch**: no *a priori* agreement on interfaces
Starting point: Cake’s big picture
Cake in one slide

Cake is

- a language expressing *compositions* of software
- a productive tool for overcoming *mismatch*
- operating on *binaries*
- designed around practical experience
- ongoing work

In this talk, I’ll cover

- two motivational case-studies
- the Cake language design
- some implementation and status
Wanted: a tool for helping with tasks like...

Unanticipated composition: port feature $P$ from app $X$ to $Y$

Case study: Konqueror + ROX-Filer

Evolution: link client version 1 against library version 2

Case study: gtk-theme-switch
Outline of the rest of this talk

- Design and first case study
- Second case study: object exchange
- The Cake language: core
- The Cake language: practicalities
- Status and questions
Experiment 1: a simple exercise in glue

I like program $X$, but it lacks feature $P$ found in program $Y$.

- let $X = \text{ROX-Filer}$, $P = \text{history}$ and $Y = \text{Konqueror}$

```c

static GList *history = NULL;  /* Most recent first */
static GList *history_tail = NULL;  /* Oldest item */

void bookmarks_add_history(const gchar *path);

GtkWidget *build_history_menu(FilerWindow *filer_window);

class LIBKONQ_EXPORT KonqHistoryManager : public KParts::HistoryProvider,
   public KonqHistoryComm {
   // ...
   void addToHistory(bool pending, const KURL& url,
      const QString& typedURL = QString::null,
      const QString& title = QString::null);

   virtual QStringList allURLs() const;
   /* ... */
};
```

Cake... – p.8/32
Decision: black-box approach

Why not hack source?

- must understand source language
- must understand code internals
- poor compositionality
- poor maintainability
- time sink?

Instead choose *black-box* approach; but possibly

- less powerful?
- performance? …
Decision: work with *binaries*

Why binaries?

- unify many source languages
- convenience
- no source code?
- debugging analogue: use DWARF metadata

*But* oblivious to

- *cross-module* macro expansion
- *cross-module* inlining
- template metaprogramming, …

(…at compile time. First two are bad ideas anyway.)
Konqueror + ROX case-study: findings

Easy:

■ converting and transferring data
■ interposing on control flow

Hard:

■ extricating the history feature from libkonq
■ altering embedded policy
■ bypassing language quirks (protected)
■ manual set-up of infrastructure state
■ understanding infrastructure (binding convention, . . . )

Interesting: difficulties were infrastructure, not application
Outline of the rest of this talk

- Design and first case study
- Second case study: object exchange
- The Cake language: core
- The Cake language: practicalities
- Status and questions
A second case study: evolution

gtk-theme-switch

- one version for Gtk+ 1.2, another for Gtk+ 2.0
- forked codebase (maintenance)
- ... diff \(-U3\) is \(\sim 500\) lines
- can one binary work with both libraries?

Main challenge: exchange of mismatched objects.
Outline of the rest of this talk

- Design and first case study
- Second case study: object exchange
- The Cake language: core
- The Cake language: practicalities
- Status and questions
Introducing the Cake language

Cake is a *configuration language*

- i.e. expresses *inter-component relationships*
- specifically: mismatch resolutions

Cake complements existing languages:

- accommodates heterogeneity
- accommodates plug-incompatibility
Cake language: basics

Two main kinds of statement:

- **exists**—describes existing binaries
- **derive**—derives new ones

A simple Cake module:

```plaintext
exists elf_reloc ("switch2.o") switch2;
exists elf_external_sharedlib ("gtk-x11-2.0") gtk-x11-2.0;
exists elf_external_sharedlib ("gdk-x11-2.0") gdk-x11-2.0;
// ... more follow
derive elf_executable ("switch2") switch-exec = make_exec(
    link [switch2, gtk-x11-2.0, gdk-x11-2.0, /* ... */]
);
```
A simple mismatch, using C++

```c
struct foo {
    int a;
    float b;
};

int manipulate_foo(struct foo *f);

struct foo {
    float b;
    int a;
    char pad_ignored[42];
};

int manipulate_foo(struct foo *f);
```

In C++, you might write

```c
int __wrap_manipulate_foo ( first :: foo *f) {
    second :: foo obj;   obj.a = f->a;   obj.b = f->b;
    int retval = second :: manipulate_foo(&obj);
    f->a = obj.a;   f->b = obj.b;
    return retval ;
}
```
A simple mismatch, in Cake

In Cake, you’d write...

...nothing! (Assuming sufficient debug information...)
Consider these two mismatched functions.

```c
#include <glib-object.h>

/* The first function */
uint gtk_signal_connect (GtkObject *,
        const gchar *,
        GtkSignalFunc f ,
        gpointer f_data);

/* The second function */
ulong g_signal_connect_data ( gpointer inst ,
        const gchar *,
        GCallback c_h ,
        gpointer data ,
        GClosureNotify destroy_data ,
        GConnectFlags flags );
```

How would you manually code around this mismatch?
In Cake:

```c
derive /* ... */ switch_exec = link[switch12, libgtk20]
{
    // ...
    switch12 ↔ libgtk20 {
        gtk_signal_connect (i, d, c_h, data)
        → g_signal_connect_data (i, d, c_h, data, null, {});
        // more correspondences ...
    }
    // more pairwise blocks...
}
```

- pattern-matching + “dual scoping”
- primitive values (e.g. guint ↔ guint) for free
Value correspondences

Name-matching gets so far. Further: *value correspondences*.

```
struct _GtkWindow {
    GtkBin bin; gchar * title;  // ...
    GtkWidgetType type;
    guint window_has_focus:1;
};
```

```
struct _GtkWindow {
    GtkWidget bin; gchar * title;  // ...
    gchar *wm_role;
    guint type :4; /* GtkWidgetType */
    guint has_focus :1;
};
```
Value correspondences

Name-matching gets so far. Further: *value correspondences*.

```c
struct _GtkWindow {
    GtkBin bin; gchar * title; // ...
    GtkWidgetType type;
    guint window_has_focus:1;
};

switch12 ← libgtk20 {
    values GtkWindow ← GtkWidget {
```
Name-matching gets so far. Further: \textit{value correspondences}.

\begin{verbatim}
struct _GtkWindow {
    GtkBin bin; gchar * title ; // ...
    GtkWidgetType type;
    guint window_has_focus:1;
};

switch12 \leftrightarrow \texttt{libgtk20} {
values GtkWidget \leftrightarrow GtkWidget {
    \texttt{void} \rightarrow \texttt{wm\_role};
}\end{verbatim}
Value correspondences

Name-matching gets so far. Further: *value correspondences.*

```
struct _GtkWindow {
    GtkBin bin; gchar * title; // ...
    GtkWidgetType type;
    guint window_has_focus:1;
};

switch12 $\leftrightarrow$ libgtk20 {
    values GtkWidget $\leftrightarrow$ GtkWidget {
        void $\rightarrow$ wm_role;
        type as GtkWidgetType $\leftarrow$ type as GtkWidgetType;
    }
}
```
Value correspondences

Name-matching gets so far. Further: value correspondences.

```c
struct _GtkWindow {
    GtkBin bin; gchar * title; // ...
    GtkWidgetType type;
    guint window_has_focus:1;
};

struct _GtkWindow {
    GtkBin bin; gchar * title; // ...
    gchar *wm_role;
    guint type:4; /* GtkWidgetType */
    guint has_focus:1;
};

switch12 ↔ libgtk20 {
    values GtkWidget ↔ GtkWidget {
        void → wm_role;
        type as .GtkWidgetType <--> type as .GtkWidgetType;
        window_has_focus ↔ has_focus;
    }
}
```
More complex correspondence patterns

Function patterns may be predicated on call content:

```c
// rule matched when using a particular argument value
gtk_type_check_object_cast (0, _) → (true);
```

Simple stub language for defining sequences (on RHS):

```c
gtk_window_set_policy (win, shrink, grow, _) →
   (if shrink then gtk_window_set_size_request (win, 0, 0) else void;
    if grow then gtk_window_set_resizable (win, TRUE) else void);
```

- future: *context*-predicated patterns (stack; sequence…)

Cake… – p.23/32
Future: interpretations of object files

In the first case study was a notion of component style.

dcop( // hide DCOP internals and enable introspection
d    kde-3.x("konqueror", // KDE initialisation and self-binding
d        qt-4.x( // import Qt initialisation constraints
d            gcc-c++-4.x( // apply name-mangling rules etc.
d                elf_reloc( // basic interpretation

d                "konq_historymgr.o"

d            )

        )

    )

) // ...

Future work needed to work all this out:

- **define** these stackable interpretations
- code generation
Outline of the rest of this talk

- Design and first case study
- Second case study: object exchange
- The Cake language: core
- The Cake language: practicalities
- Status and questions
How Cake understands object files

Cake gets quite far using debugging info. But can also use:

- annotations in `exists` blocks
  - to supplement debugging info
  - to enable optimisations
- static analysis (none yet...)
```c
exists elf_reloc ("switch.o") switch12 {
    declare {
        gtk_dialog_new : _ ⇒ object { // function returning a
            vbox: opaque ptr;             // pointer to an object, where
            _ : ignored;                  // vbox is opaque to switch12,
        } ptr                             // and other fields are ignored
    } /* more annotations ... */ }
```

- **opaque** and **ignored** used during object exchange...
- ... to limit depth of deep copy
Choose your own adventurousness

Annotations can be made with varying strength.

- **if check**, annotation must be verifiable
  - from metadata or static analysis
- **if declare**, annotations must not contradict metadata
- **if override**, contradiction is allowed

Composition tasks naturally cover the spectrum...

```c
exists  elf_reloc ("switch.o") switch12 {
    override  { gtk_dialog_new : _ ⇒ GtkWidget *; }
    /* static type in source code is imprecise (GtkWidget) */
}
```
Cake describes treatment of values, not types

- there is no type system in Cake (but could be added)

**But** the Cake compiler consumes static metadata!

- necessary: can’t assume RTTI, e.g. in C
- need stronger assumptions e.g. than C
- imprecise static types cause problems

Primitive “data types” (i.e. value forms) are defined by a pair

- DWARF *encoding* (e.g. unsigned, float, fixed, …)
- length in bytes

From these, get pointers, enums, structures, …
Summary: what Cake gives the programmer

Cake \textit{relates} heterogeneous, mismatched components.

- zero-effort handling of simple binary incompatibilities
- expressive pattern-matching
- allow conservative or relaxed coding
- convenient pairwise correspondences ("dual scope")
- flexible treatment of data encoding and identifiers
- (in future) abstraction over component styles
Outline of the rest of this talk

- Design and first case study
- Second case study: object exchange
- The Cake language: core
- The Cake language: practicalities
- Status and questions
Status

Case-studies implemented as hand-written glue code...

- *but* Gtk case study is partially automated
- stubs and conversion functions are generated by scripts

Cake compiler is ongoing right now

- parses, reads DWARF, merges annotations, complains
- more soon

Runtime library `rep_man` is most developed piece

- used in the Gtk case study
- completely generic

Thanks for your attention. Any questions?